digital whisper
Technologic Papers

Linux kernel so injector

מאת אדיר מקמל

הקדמה

בשלושת השנים האחרונות עסקתי בעיקר בפיתוח ומחקר low-level במערכות הפעלה מודרניות.

בתקופה האחרונה רציתי ללמוד ולשפר את יכולת הפיתוח הקרנלי שלי בלינוקס על מנת להבין טוב יותר את התקשורת עם החומרה וכמובן גם בשביל להבין את המנגנונים השונים בקרנל.

בחרתי דווקא בלינוקס מכיוון שלינוקס זאת מערכת הפעלה שבאה עם קוד פתוח כלומר, הקרנל של לינוקס וגם התוכנות של ההפצות השונות מגיעות כשהקוד מקור שלהן מפורסם ולרוב גם מתועד (ברמה כזו או אחרת).

על מנת להקל על הפיתוח בקרנל הייתי צריך להרים סביבה משלי שהיה לי נוח לקמפל, לדבג ולפתח. השתמשתי בפרויקט buildroot על מנת לקמפל קרנל מינימאלי לצרכים שלי וליצור rootfs מתאים. השתמשתי ב-VSCODE כסביבת הפיתוח וקנפגתי השלמה אוטומטית לקבצי מקור של הקרנל (יש לינק בסוף ל-repo של הסביבה).

במאמר זה אציג PoC שעשיתי להזרקת ספריות משותפות מ-kernel mode ל-user mode על ידי דרייבר שכתבתי.

ה-PoC פותח ונבדק על קרנל גרסה: 4.19.91. בנוסף לכך, קיימת תלות ב-glibc שעליה יורחב בהמשך.

לפני שנמשיך, חשוב לי להבהיר שהקוד מוגדר כ-PoC ואינו מטפל בכל מקרי הקצה האפשריים. אנחנו נראה לא מעט חלקים ב-PoC שהיו יכולים להיעשות טוב יותר ולהיכתב בצורה שתוביל לפחות באגים.

הגדרות ומושגים

בחלק זה נסביר בקצרה על קרנל, signals ו-glibc (אם הנכם מכירים את הנושאים האלו אני ממליץ לדלג לחלק הבא).

Kernel

ישנם כמה סוגים של קרנלים, העיקריים הם: microkernel ו-monolithic kernel. הקרנל של לינוקס הוא monolithic kernel (מעתה והילך אנחנו נדון רק על monolithic kernel). הקרנל הוא החלק במערכת ההפעלה שאחראי על ניהול תהליכים, ניהול זיכרון, טיפול ב-syscalls וכמובן תקשורת עם החומרה במחשב. על מנת לצמצם את הקוד שרץ במרחב הכתובות הקרנלי, הקרנל תומך בטעינה דינאמית של דרייברים בהתאם לחומרה הקיימת במחשב. לא מדובר בדרייברים צד שלישי אלא בדרייברים עצמם של מערכת ההפעלה (ניתן לקמפל את הקרנל גם כבינארי אחד שהדרייברים מקומפלים לתוכו).

בשתי התצורות שהקרנל של לינוקס יכול להופיע בהן, ניתן בדרך כלל להוסיף דרייברים צד שלישי, כלומר קטע קוד אחר שירוץ בקרנל שיכול להיכתב על ידי כל מפתח (לפעמים נצטרך חתימה דיגיטלית על מנת לטעון דרייבר, אך במאמר זה לא נתעסק במקרה זה).

Signals

בלינוקס ישנן כמה דרכים לתקשר בין תהליכים, סיגנלים הוא חלק מהן. סיגנלים הם הודעות הנשלחות מהקרנל או מתהליך לתהליך ספציפי, סיגנל נשלח לתהליך כדי להודיע לו על אירוע שהתרחש. דוגמה למספר סיגנלים נפוצים: SIGSTOP, SIGKILL, SIGCONT ו-SIGSEGV.

תהליך יכול לרשום callback לסיגנל בעזרת sigaction syscall. ניתן לרשום callback לכל סיגנל חוץ משניים מיוחדים: SIGKILL ו-SIGSTOP. כלומר, ניתן לעצור ו-"להרוג" כל תהליך במערכת ההפעלה (חוץ מ-init שאותו לא ניתן "להרוג" גם אם נשלח בטעות או בכוונה SIGKILL, הקרנל לא יעביר את הסיגנל אליו). עוד מידע ניתן למצוא ב-man 7 signals.

Glibc

כל C compiler מממש את הספרית C הסטנדרטית. לדוגמה, הקומפיילר MSVC של מיקרוסופט, מממש את VCRUNTIME. במקרה שלנו מדובר ב-GCC של GNU שמממש את glibc. glibc היא הספרית C הסטנדרטית שפותחה על ידי GNU. הספרייה מספקת עיבוד קלט ופלט, התעסקות עם מחרוזות, ניהול זיכרון והכי חשוב עטיפה לקריאות מערכת.

שלבי הזרקת הספרייה

על מנת להזריק את הספרייה המשותפת נצטרך מזהה של תהליך ואת הנתיב לספרייה. הזרקת הספרייה המשותפת מתחלקת לשלבים הבאים:

1. שליחת SIGSTOP לתהליך יעד.
2. מציאת מקום פנוי ל-shellcode שלנו (כל כתובת שיש לה הרשאות של קריאה והרצה R^X).
3. מציאת הספרייה glibc אשר טעונה לתהליך יעד.
4. פרסור glibc elf בזמן ריצה על מנת למצוא את הכתובת של libc_dlopen_mode.
5. השגת אוגר rip של תהליך היעד.
6. בניית ה-shellcode שלנו עם הכתובות הנכונות של:
   a. הכתובת שאליה נכתב הנתיב לספרייה המשותפת שנרצה לטעון.
   b. אוגר rip של התהליך על מנת שנוכל לשחזר את הריצה של התהליך אחרי שטען את הספרייה שלנו.
   c. הכתובת של libc_dlopen_mode שאליה נרצה לקפוץ על מנת לטעון את הספרייה.
7. כתיבת הנתיב לספרייה המשותפת למרחב הכתובות של תהליך היעד.
8. כתיבת ה-shellcode למרחב הכתובות של התהליך.
9. שינוי אוגר rip של התהליך שיצביע לכתובת של ה-shellcode שלנו.
10. שליחת SIGCONT לתהליך יעד.

שלבים 1-3

בשלב הראשון נשלח לתהליך היעד את הסיגנל בעזרת הפונקציה send_sig.

```
60      // stop the target process
61      status = send_sig(SIGSTOP, target_task, KERNEL_PRIV);
62      if (0 > status) {
63          printk(KERN_INFO "Unable to stop the process, pid %d\n", parameters->pid);
64          return SIGSTOP_FAILED;
65      }
66      is_stopped = true;
67      printk(KERN_INFO "The process stopped successfully, pid %d\n", parameters->pid);
```

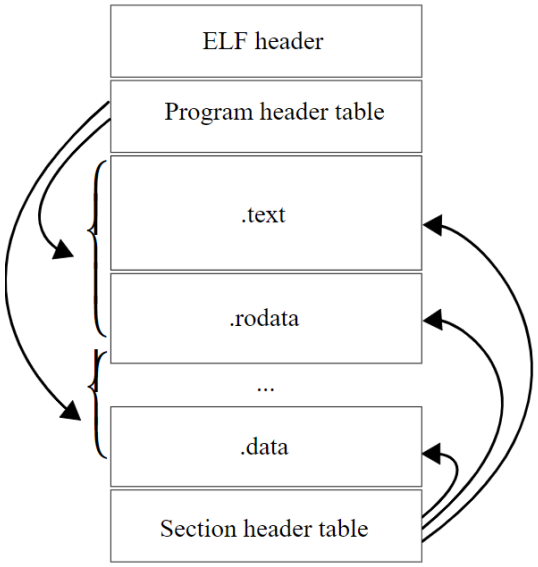
לאחר מכן, נקרא את /proc/pid/maps על מנת למצוא זיכרון בתהליך שהוא עם הרשאות קריאה והרצה (מהקרנל ניתן לכתוב לדפים של תהליכים גם כאשר הדפים בלי הרשאות כתיבה). כמו כן, נשתמש באותה שיטה כדי למצוא את הכתובת של glibc.

על מנת לקרוא ולכתוב מקבצים, הקרנל מייצא לדרייברים עטיפה ל-VFS שבה נשתמש. בנוסף לכך, חיפשתי דרך נוחה לקרוא ולכתוב מזיכרון של תהליכים ומצאתי שהכי נוח יהיה להשתמש בפונקציה

mem_rw מ-proc fs מפני שזאת תכליתה ולעומת קוד שאחרים כתבו בפורומים שונים או קוד שאני אכתוב הסיכוי לבאגים ב-mem_rw נמוך יותר. בגלל שהפונקציה הזאת לא מיוצאת נאלץ להעתיק ולשנות אותה על מנת שתעבוד עם buffers שנמצאים במרחב הכתובות הקרנלי.

שלב 4

בשלב זה יש לנו את השם של הפונקציה שאת כתובתה נרצה למצוא (libc_dlopen_mode) וכמו כן גם את כתובת ההתחלה של glibc. בעזרת שני פרמטרים אלו נוכל לפרסר את מבנה ה-elf של glibc כשהוא טעון בזיכרון ולמצוא את הכתובת הדינאמית של libc_dlopen_mode.



[מקור]

בתמונה משמאל ניתן לראות תיאור כללי של ELF format, לפי התמונה בכתובת הראשונה מתחיל ה- ELF header שבמיקום מסוים ממנו נמצא Program header table ומשם ניתן להגיע ל-sections השונים.

כך ELF header נראה:

```
220
221  typedef struct elf64_hdr {
222    unsigned char e_ident[EI_NIDENT]; /* ELF "magic number" */
223    Elf64_Half e_type;
224    Elf64_Half e_machine;
225    Elf64_Word e_version;
226    Elf64_Addr e_entry;   /* Entry point virtual address */
227    Elf64_Off e_phoff;    /* Program header table file offset */
228    Elf64_Off e_shoff;    /* Section header table file offset */
229    Elf64_Word e_flags;
230    Elf64_Half e_ehsize;
231    Elf64_Half e_phentsize;
232    Elf64_Half e_phnum;
233    Elf64_Half e_shentsize;
234    Elf64_Half e_shnum;
235    Elf64_Half e_shstrndx;
236  } Elf64_Ehdr;
```

לא נתייחס לכל ה-struct, אלא רק לשדה שמעניין אותנו שהוא e_phoff - המיקום של Program header table מתחילת ה-ELF header.

Program header table הוא מערך של מבנים שכל מבנה בו מכיל מידע שהמערכת או ה-loader זקוקים לו כדי להכין את התהליך או הספרייה לריצה. המבנה שמעניין אותנו הוא PT_DYNAMIC (לפי p_type נוכל לדעת איזה מבנה זה) שמכיל מערך של מבנים אשר מכילים מידע של קישור דינאמי אשר מחושב בזמן טעינת ה-ELF.

על מנת למצוא את ה-dynamic section:

```
int get_dynamic_section(struct task_struct* task, unsigned long number_of_entries,
 unsigned long module_base_address,unsigned long offset, unsigned long* dynamic_tables) {

    ssize_t result;
    Elf64_Phdr elf_section;
    size_t i;
    for (i = 0; i < number_of_entries; i++) {
        result = mem_read(task, (char*)&elf_section, sizeof(Elf64_Phdr), offset +
                                                        (sizeof(Elf64_Phdr) * i));
        if (sizeof(Elf64_Phdr) != result) {
            return GET_DYNAMIC_SECTION_FAILED;
        }
        if (PT_DYNAMIC == elf_section.p_type) {
            *dynamic_tables = module_base_address + elf_section.p_vaddr;
            return SUCCESS;
        }
    }
    return GET_DYNAMIC_SECTION_FAILED;
}
```

לפי man 5 elf:

```
Dynamic tags (Dyn)
    The .dynamic section contains a series of structures that hold rele-
    vant dynamic linking information.  The d_tag member controls the
    interpretation of d_un.

        typedef struct {
            Elf32_Sword    d_tag;
            union {
                Elf32_Word d_val;
                Elf32_Addr d_ptr;
            } d_un;
        } Elf32_Dyn;
        extern Elf32_Dyn _DYNAMIC[];

        typedef struct {
            Elf64_Sxword    d_tag;
            union {
                Elf64_Xword d_val;
                Elf64_Addr  d_ptr;
            } d_un;
        } Elf64_Dyn;
        extern Elf64_Dyn _DYNAMIC[];
```

מתוך הטבלאות הנמצאות ב-dynamic section, שתי הטבלאות שמעניינות אותנו הן: DT_STRTAB ו-DT_SYMTAB.

DT_SYMTAB מייצגת את הטבלה של הסימבולים ו-DT_STRTAB מייצגת את הטבלה של השם של כל סימבול בהתאמה:

```
void* find_symbol(struct task_struct* task, unsigned long symbol_string_table, unsigned long symbol_table,
const char* symbol_name) {
    Elf64_Sym symbol;
    ssize_t result;
    size_t i = 0;
    char found_symbol_name[MAX_SYMBOL_NAME] = { 0 };
    size_t symbol_len = strlen(symbol_name);
    while (true) {
        result = mem_read(task, (char*)&symbol, sizeof(Elf64_Sym), symbol_table + (sizeof(Elf64_Sym) * i));
        if (sizeof(Elf64_Sym) != result) {
            return NULL;
        }
        result = mem_read(task, found_symbol_name, symbol_len, symbol_string_table + symbol.st_name);
        if (symbol_len != result) {
            return NULL;
        }
        if (0 == strncmp(symbol_name, found_symbol_name, symbol_len)) {
            return (void*)symbol.st_value;
        }
        ++i;
        memset(found_symbol_name, 0, MAX_SYMBOL_NAME);
    }
    return NULL;
}
```

למה אנחנו צריכים את libc_dlopen_mode?

על מנת לטעון ספרייה משותפת בזמן ריצה, ניתן להשתמש בספרייה dl אך לא כל תהליך משתמש בספרייה זו ולכן לא נוכל להשתמש בה. למזלנו, glibc היא מהספריות הנפוצות שיש ורוב ההפצות הלינוקסיות משתמשות בה, לא כל תהליך חייב להשתמש בספרייה זו.

libc_dlopen_mode היא המימוש של libc ל-dlopen אשר משמשת על מנת לטעון ספרייה דינאמית לתהליך בזמן ריצה.

שלבים 5-9

הקוד המתואר בתמונה הבאה יחסית פשוט, בודקים אם התהליך שאליו מזריקים היה במהלך syscall. לאחר מכן, ממשיכים על מנת לבנות את ה-shellcode (עוד מעט נצלול לתוכו), כותבים את הנתיב של הספרייה המשותפת למרחב הכתובות של התהליך בכתובת שמצאנו לפני. לאחר שלבים אלו, כותבים את ה-shellcode עצמו למרחב הכתובות של התהליך.

```
// did we stopped from a system call?
if  (syscall_get_nr(target_task, task_pt_regs(target_task)) >= 0){
    came_from_syscall = true;
}
else {
    came_from_syscall = false;
}

// get the shellcode parsed and patched to correct addresses
shellcode = get_shellcode(&shellcode_size, task_pt_regs(target_task), (unsigned long)free_addr,
(unsigned long)symbol_address, came_from_syscall);

if (NULL == shellcode) {
    printk(KERN_INFO "Unable to get the shellcode\n");
    goto release_process;
}

// write to so path to process memory include the null-terminator
if(parameters->so_path_size != mem_write(target_task, parameters->so_path, parameters->so_path_size,
(unsigned long)free_addr)) {
    printk(KERN_INFO "Unable to write the so path to process memory, pid %d\n", parameters->pid);
    goto release_shellcode;
}

// write to so path to process memory include the null-terminator
if(shellcode_size != mem_write(target_task, shellcode, shellcode_size,  (unsigned long)free_addr +
parameters->so_path_size)) {
    printk(KERN_INFO "Unable to write the shellcode to process memory, pid %d\n", parameters->pid);
    goto release_shellcode;
}

// did we stopped from a system call?
if  (came_from_syscall){
    task_pt_regs(target_task)->ip = (unsigned long)free_addr + parameters->so_path_size + 2;
}
else {
```

לפני שצלול ל-shellcode צריך להבין את המצבים השונים שבהם תהליך יכול להיות כאשר נעצר על ידי SIGSTOP וחוזר לרוץ על ידי SIGCONT. סיגנלים נשלחים על ידי הקרנל בזמן החזרה מ-kernel mode ל-user mode (syscalls, context switch etc).

נשאלת השאלה מה הקרנל יעשה כאשר SIGSTOP נשלח בזמן שקריאת מערכת הייתה חסומה. דוגמה לכך, בזמן שתהליך שלנו ביצע open לקובץ בו-זמנית נשלח מהקרנל או מתהליך אחר SIGSTOP לתהליך שלנו. כלומר, התהליך שלנו יעצור בזמן ביצוע ה-syscall. אך מה יקרה כאשר נשלח SIGCONT הרי התהליך היה באמצע ביצוע של syscall.

ישנן שתי אופציות שבהן הקרנל יכול לבחור והן בהתאם לסוג ה-syscall שהיה בביצוע:

1. אחרי שה-signal handler יסתיים הקרנל ישנה את האוגרים כך ששוב ה-syscall יתבצע.
2. הקרנל יכשיל את הקריאה עם ערך שגיאה EINTR.

בואו נראה את הקוד בקרנל שמטפל בזה (הקוד שמטפל בביצוע של סיגנלים הוא תלוי ארכיטקטורה, בחרתי להראות דוגמה מ-x86):

```
static void
handle_signal(struct ksignal *ksig, struct pt_regs *regs)
{
        bool stepping, failed;
        struct fpu *fpu = &current->thread.fpu;
        if (v8086_mode(regs))
                save_v86_state((struct kernel_vm86_regs *) regs, VM86_SIGNAL);
        /* Are we from a system call? */
        if (syscall_get_nr(current, regs) >= 0) {
                /* If so, check system call restarting.. */
                switch (syscall_get_error(current, regs)) {
                case -ERESTART_RESTARTBLOCK:
                case -ERESTARTNOHAND:
                        regs->ax = -EINTR;
                        break;
                case -ERESTARTSYS:
                        if (!(ksig->ka.sa.sa_flags & SA_RESTART)) {
                                regs->ax = -EINTR;
                                break;
                        }
                        fallthrough;
                case -ERESTARTNOINTR:
                        regs->ax = regs->orig_ax;
                        regs->ip -= 2;
                        break;
                }
        }
```

בזמן הזרקת הספרייה, הקרנל יגיע פעמיים לפונקציה handle_signal, פעם אחת כשנשלח SIGSTOP ופעם אחת כשנשלח SIGCONT.

עבור הפעם הראשונה כשמדובר ב-SIGSTOP הקרנל יבדוק אם הוא היה במהלך syscall. אם כן, בהתאם לסוג ה-syscall הקרנל יבחר אם לבצע אתחול מחדש של ה-syscall על ידי חיסור rip ב-2 (הגודל של syscall opcode) או להחזיר EINTR כלומר להחזיר לתהליך שה-syscall בוטל. בפעם השנייה הקרנל יגיע לפונקציה זו כשנשלח SIGCONT על מנת להריץ את התהליך שוב לאחר שנעצר, בפעם הזאת התהליך בהכרח לא היה באמצע syscall לכן הקרנל לא יכנס ל-if הזה.

לכן, אחרי שאנחנו מבצעים את ההזרקה ועוצרים את התהליך, אנחנו נצטרך להתחשב במצב שבו עצרנו את התהליך בזמן ביצוע ה-syscall. ישנם לא מעט syscalls שלאחר ביצוע SIGSTOP ואז SIGCONT לא יאותחלו מחדש והקרנל יחזיר EINTR, בשביל לראות את הרשימה המלאה man 7 signal.

על מנת לבנות את ה-shellcode שלנו נצטרך שלושה פרמטרים:

1. את הכתובת של הפונקציה שמצאנו של libc_dlopen_mode.
2. את הכתובת שאליה כתבנו את הנתיב של הספרייה הדינאמית שאותה נרצה לטעון.
3. את הכתובת שאליה נרצה לחזור, כלומר התהליך היה בכתובת מסוימת כשעצרנו אותו ועל מנת לשחזר את הריצה של התהליך מבלי לגרום לו לקרוס נצטרך לחזור לכתובת זו. כפי שהזכרנו למעלה, נצטרך לבדוק אם התהליך היה בעת ביצוע syscall ואם כן נחסיר את הכתובת יעד שאליה נרצה לקפוץ בגודל ה-opcode של syscall על מנת שיתבצע שוב (בעולם אידיאלי היינו בודקים שזה syscall שתומך ב-restart בדיוק כמו שהקרנל מבצע על מנת לשחזר את הריצה).

השחזור של האוגרים לא מושלם מפני שאוגר r15 לא משתחזר.

לא לשכוח שהמחסנית חייבת להיות aligned ל-16 כשקוראים ל- mode_dlopen_libc.

```
22          pushq %rax
23          pushq %rbx
24          pushq %rcx
25          pushq %rdx
26          pushq %rdi
27          pushq %rsi
28          pushq %rsp
29          pushq %rbp
30          pushq %r8
31          pushq %r9
32          pushq %r10
33          pushq %r11
34          pushq %r12
35          pushq %r13
36          pushq %r14
37          pushq %r15
38          pushfq
39
40          movq %rsp, %rbx
41          andq $0xFFFFFFFFFFFFFFF0, %rsp
42
43          movq $MAGIC_SO_LOAD_FUNCTION, %rax
44          movq $MAGIC_SO_PATH, %rdi
45          mov $(RTLD_NOW | __RTLD_DLOPEN), %rsi
46          callq *%rax
47
48          movq %rbx, %rsp
49
50          popfq
51          popq %r15
52          popq %r14
53          popq %r13
54          popq %r12
55          popq %r11
56          popq %r10
57          popq %r9
58          popq %r8
```

```
59          popq %rbp
60          popq %rsp
61          popq %rsi
62          popq %rdi
63          popq %rdx
64          popq %rcx
65          popq %rbx
66          popq %rax
67
68          movq $MAGIC_PREV_IP, %r15
69          jmpq *%r15
```

מצטער על ה-AT&T syntax...

אחרי שבנינו את ה-shellcode עם הכתובות הנכונות נכתוב את ה-shellcode למרחב הכתובות של התהליך בכתובת שמצאנו לפני שיש לה את ההרשאות המתאימות (R^X).

לאחר מכן, נרצה לשנות את אוגר rip שיצביע לכתובת של ה-shellcode שלנו. חשוב לזכור, במצב שבו נשנה את rip להצביע ל-shellcode שלנו אך לא נתחשב בעובדה שהתהליך עלול היה לעצר בזמן ביצוע syscall, יכול להיווצר מצב שכאשר נשלח SIGCONT לתהליך, הקרנל עלול לשנות את הכתובת שאליה יחזור התהליך לכתובת ששמנו באוגר rip-2 (כמו שהוזכר למעלה פחות הגודל של syscall opcode). לכן נעשית בדיקה אם התהליך חזר מביצוע syscall אז נעלה את rip ב-2 ולאחר מכן הקרנל יוריד ב-2 ונגיע בדיוק לכתובת של ה-shellcode.

שלב 10

בשלב זה נשלח לתהליך היעד SIGCONT על מנת שיחזור לרוץ (אם כל השלבים עברו בהצלחה התהליך יחזור לרוץ מה-shellcode) ולטעון את הספרייה הדינאמית שרצינו להזריק לו.

```
130     // continue the target process in order to execute our so shellcode loader
131     send_sig(SIGCONT, target_task, KERNEL_PRIV);
132     is_stopped = false;
```

מגבלות ידועות

1. כרגע אין תמיכה בקוד ב-syscalls מהסוג שחייב להחזיר להם EINTR בחזרה כאשר נעצרים על ידי SIGSTOP. או ב-syscalls שהיו צריכים להיקרא בעזרת restart_syscall אחרי שנעצרו על SIGSTOP וחזרו לרוץ אחרי SIGCONT.
2. כרגע קיים מימוש רק לארכיטקטורת x86_64.
3. הקוד מוגדר כ-PoC ולא נכתב עם טיפול מספיק נכון בשגיאות ולא עבר מספיק בדיקות ולכן באגים יכולים לקרות במקרי קצה מסוימים.
4. ה-shellcode והנתיב לספרייה הדינאמית אינם מתנקים. כלומר, לאחר הכתיבה לתהליך והרצת ה-shellcode הדבר הנכון לעשות הוא לשחזר את המידע שהיה שם לפני גם אם הוא לא בשימוש. לא היה לי מספיק זמן לממש את זה ב-kernel thread נפרד אבל יש commit שעושה את זה באותו ה-thread. בחרתי להוריד את ה-commit הזה ב-branch הראשי בגלל שתחת מעבד אחד הוא "יתקע" את הקרנל כלומר את המחשב.
5. אוגר r15 לא משתחזר.

חשוב לציין שכל המגבלות שצוינו מעל הינן פתירות.

סיכום

אז מה עשינו? עצרנו תהליך מסוים שבחרנו, פרסרנו בעזרת /proc/pid/maps היכן בתהליך יש זיכרון R^X וגם מה הכתובת של הספרייה glibc שנטענה אל התהליך.

לאחר מכן, פרסרנו את glibc בזיכרון כלומר את הפורמט של elf, על מנת למצוא את הפונקציה/סימבול libc_dlopen_mode.

התחלנו לבנות את ה-shellcode שלנו שבהמשך יטען את הספרייה המשותפת על ידי כך שיקרא ל-libc_dlopen_mode. כל מה שה-shellcode שלנו צריך זה: הכתובת שאליה נכתוב את הנתיב לספרייה הדינאמית שנרצה לטעון, הכתובת אליה נקפוץ על מנת לטעון את הספרייה כלומר libc_dlopen_mode והכתובת שאליה נחזור על מנת לשחזר את הריצה של התהליך.

לאחר מכן, כתבנו את המחרוזת שמייצגת את הנתיב לספרייה הדינאמית ואת ה-shellcode עצמו למרחב הכתובות של התהליך.

שינינו את אוגר rip של התהליך שיצביע לכתובת שאליה כתבנו את ה-shellcode ולבסוף שלחנו לתהליך SIGCONT על מנת שיחזור לרוץ מה-shellcode שלנו.

- ה-repo של הפרויקט: https://github.com/Rhydon1337/linux-kernel-so-injector
- ה-repo של הסביבה: https://github.com/Rhydon1337/linux-kernel-development

על המחבר

- **אדיר מקמל (Rhydon1337)**, 20, חוקר אבטחת מידע ומפתח.
  - אימייל: adirmak1@gmail.com
  - GitHub: https://github.com/Rhydon1337